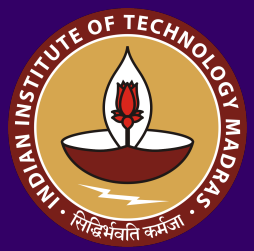
Scientific Computing & Advanced Programming Lab

SAURAV SAMANTARAY

Department of Mathematics
IIT Madras

Lecture-Slides-3



sauravsray@iitm.ac.in

From Data Structures to Objects

- So far, we have seen different ways of organising data.
- For example, a point can be represented using a tuple: $P = (3, 4)$
- We can store the data, but what about operations associated with that data?
- For a point, we might want to:
  - Find its distance from the origin.
  - Find its distance from another point.
  - Translate the point.
  - Find its polar coordinates.
- Instead of keeping the **data** and the **functions** separate, we can combine them into a single entity.

$$\boxed{\text{Data} + \text{Operations} \longrightarrow \text{Object}}$$

Introduction to Object-Oriented Programming

- **Object-Oriented Programming (OOP)** organizes a program around **objects**.
- An object contains:
  - **Attributes** — data describing the object.
  - **Methods** — operations that the object can perform.

Example: A Point

Point
$x,\ y$
distance() translate() polar_coordinates()

- x and y are the **attributes**,
- the functions are the **methods**.

The object therefore combines the **state** of an entity with the **operations** that act on that state.

Classes and Objects

- A **class** is a blueprint for creating objects.
- An **object** is a particular instance of a class.
- The class defines what a `Point` contains and what it can do.
- We can then create different objects:

```
P = Point(3, 4)
Q = Point(7, 2)
```

Point class $\longrightarrow$ blueprint

$\downarrow$

$P = (3, 4)$

$Q = (7, 2)$

One class $\longrightarrow$ many objects

A First Mathematical Class: `Point`

- We want a Python object to represent a point $P = (x, y)$.
- The point should store its coordinates and calculate its distance from the origin.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return (self.x**2 + self.y**2)**0.5
```

The class combines:

$\underbrace{x, y}_{\text{data}}$ + $\underbrace{\text{distance\_from\_origin()}}_{\text{operation}}$

Using the `Point` Class

- The class is a blueprint. We can create different **objects** from it.

```
P = Point(3, 4)
Q = Point(6, 8)
```

- Each object has its own coordinates:

```
print(P.x, P.y)          # 3 4
print(Q.x, Q.y)          # 6 8
```

- The same method works on different objects:

```
print(P.distance_from_origin())    # 5.0
print(Q.distance_from_origin())    # 10.0
```

One class $\longrightarrow$ many independent objects

Dissecting the `Point` Class

```
class Point:
```

- Defines a new type called `Point`.

```
def __init__(self, x, y):
    self.x = x
    self.y = y
```

- `__init__` initializes a new object.
- `self` refers to the particular object.
- `x` and `y` are input parameters.
- `self.x` and `self.y` are object attributes.

```
def distance_from_origin(self):
    return (self.x**2 + self.y**2)**0.5
```

- A function defined inside a class is called a **method**.
- `self.x` and `self.y` refer to the coordinates of the

Functions vs. Methods

- A **function** is defined independently and receives the data it needs as arguments.

Ordinary Function

```
def distance(P, Q):
    dx = P[0] - Q[0]
    dy = P[1] - Q[1]
    return (dx**2 + dy**2)**0.5
P = (2, 3)
Q = (8, 7)
d = distance(P, Q)
```

- The same function is called independent of the points

Functions vs. Methods

- A **method** is a function defined inside a class and is called on an object.

Class Method

```
P = Point(2, 3)
Q = Point(8, 7)

d = P.distance_to(Q)
```

Function: `distance(P,Q)`

Method: `P.distance_to(Q)`

- **Key difference:** A method is associated with an object and can directly access that object's attributes through `self`.

Exercise: Extend the `Point` Class

- We have the `point` class defined as before

Task: Add the following features

1. Add a method `distance_to(other)` that calculates the distance between two points.
2. Add a method `translate(dx,dy)` that moves the point by
$$(x, y) \longrightarrow (x + dx, y + dy).$$
3. Add a method `midpoint(other)` that returns the midpoint between two points.

Test your class with

```
P = Point(2, 3)
Q = Point(8, 7)
```

Printing an Object

- Suppose we create a `Coordinate` object:

```
c = Coordinate(3, 4)
print(c)
```

- By default, Python produces something like:

```
<__main__.Coordinate object at 0x7fa918510488>
```

- This representation is not very informative!

How can we control what is displayed?

- Define a special `__str__` method for the class.
- Python automatically calls `__str__` when the object is used with `print()`.
- We decide what information should be displayed.

Extend the Point Class

- For example, we may want to add a method to display in the coordinate format (a, b).

to produce

```
print(c)
```

```
(3, 4)
```

```
def __str__(self):
    return "(" +str(self.x) + "," +str(self.y) + ")"
```

What is happening?

- `__str__` is a special method that defines the **string representation** of an object.
- `str(...)` converts the numerical coordinates into strings.
- The `+` operator joins the strings together.
- `print(P)` automatically calls `P.__str__()`.

How Does `print()` Use `__str__`?

- There is only one built-in `print()` function.
- When Python executes

```
print(P)
```

  it asks the object for its string representation:

```
P.__str__()
```

- If the class defined, then Python uses this method automatically.

```
print(P) ⟶ P.__str__() ⟶ "(3,4)"
```

Key idea: `__str__` tells Python how to represent objects of your class as a string.

OOP Exercise: Solving a 2×2 Linear System

Consider the linear system

$$a_{11}x + a_{12}y = b_1,$$
$$a_{21}x + a_{22}y = b_2.$$

For example:

- Design an object-oriented program to solve the system.
- **Partition the problem into 2–3 classes.**

- `Matrix` — stores the coefficient matrix and provides basic matrix operations.
- `Vector` — stores the right-hand side and solution vectors.
- `LinearSystem` — combines the matrix and vector and implements a method to solve

$$A\mathbf{x} = \mathbf{b}.$$

OOP Exercise: Solving a 2×2 Linear System

Test your solver with

$$\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}.$$

Expected solution:

$$\boxed{x = \frac{9}{5}, \qquad y = \frac{7}{5}}$$

Object-Oriented Design

`Matrix`

- Store matrix entries
- Access elements
- Basic matrix operations

`Vector`

- Store vector entries
- Access elements
- Basic vector operations

`LinearSystem`

- Store A and $\mathbf{b}$
- Solve

$$A\mathbf{x} = \mathbf{b}$$

- Return solution $\mathbf{x}$

`LinearSystem` $\longrightarrow$ `Matrix` + `Vector`

Design principle

Each class should have a clear responsibility:

Class Attributes

- A **class attribute** belongs to the class rather than to a particular object.

```
class Student:
    university = "IIT"

    def __init__(self, name):
        self.name = name
s1 = Student("Alice")
s2 = Student("Bob")
print(s1.name)          # Alice
print(s2.name)          # Bob
print(s1.university)    # IIT
print(s2.university)    # IIT
```

Class Attributes

Here:

`name` → object attribute

`university` → class attribute

All objects share the same class attribute:

```
Student.university
```

Key idea

Use a class attribute when a value is common to all objects of the class.

Multiple Objects: Representing a Polygon

- A polygon can be represented by a collection of `Point` objects.

```
vertices = [ Point(0, 0), Point(4, 0), Point(5, 3),
    Point(2, 5), Point(-1, 3)]
```

We can then process the collection of objects:

```
for p in vertices:
    print(p)
```

For example, compute the perimeter:

```
perimeter = 0

for i in range(len(vertices)):
    p1 = vertices[i]
    p2 = vertices[(i + 1) % len(vertices)]

    dx = p2.x - p1.x
    dy = p2.y - p1.y

    perimeter += (dx**2 + dy**2)**0.5

print(perimeter)
```

Multiple objects $\longrightarrow$ collection of related objects

Direct Access to Object State

- Consider the `Point` class defined earlier:

```
p = Point(2, 3)
p.x = 10
p.y = -50
```

- Python allows the attributes to be changed directly.
- But suppose our application requires

$$0 \leq x \leq 10, \qquad 0 \leq y \leq 10.$$

```
p.y = -50
```

has introduced an invalid state.

Problem

Any part of a large program can accidentally modify the object's data and leave the object in an invalid state.

Controlling Access with Getters and Setters

- We can provide methods for accessing and modifying the coordinates.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_y(self):
        return self.y

    def set_y(self, y):
        if 0 <= y <= 10:
            self.y = y
        else:
            print("Invalid y-coordinate")
```

Now the program uses the setter:

```
p.set_y(7)       # accepted
p.set_y(-50)     # rejected
```

`get_y()` → read

`set_y(y)` → controlled modification

Protecting the Object's State

- Instead of allowing the data to be changed directly:

```
p.y = -50          # potentially dangerous
```

- we provide a controlled interface:

```
p.set_y(-50)
```

- The setter decides whether the change is allowed.

Outside code $\longrightarrow$ `set_y()` $\longrightarrow$ validation $\longrightarrow$ object state

Key idea

- The class takes responsibility for maintaining the validity of its own state.
- This reduces the possibility of accidental or invalid

Information Hiding in Python

- Consider the `Point` class defined earlier:

```
p = Point(2, 3)
```

- Python allows us to directly access its attributes:

```
print(p.x)
print(p.y)
```

- We can also change them from outside the class:

```
p.x = 100
```

- We can even create a new attribute:

```
p.color = "red"
```

even though `color` was never defined in `Point`.

Information Hiding in Python

Good Practice

- Avoid directly manipulating the internal state of an object.
- Use methods provided by the class:

```
p.get_x()
p.set_x(5)
```

Python allows it, but good design discourages it.

OOP Problem: Polynomial Root Finding

Consider a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

Task 1: Design a class `Polynomial` to represent a polynomial. The class should be able to:

- store the coefficients,
- evaluate $p(x)$ for a given x,
- compute its derivative,
- provide a readable representation of the polynomial.

For example, the polynomial $p(x) = 2x^3 - 4x + 1$ could be represented by $[1, -4, 0, 2]$.

OOP Problem: Finding a Root

- Now consider the problem of solving $p(x) = 0$ for $p(x) = x^3 - 4x - 9$

Task 2: Design a class `RootFinder` that finds an approximate root of a given polynomial. The class should:

- take a `Polynomial` object,
- use an initial guess,
- iteratively search for a root,
- stop when the required accuracy is reached,
- report the approximate root.

`Polynomial` $\longrightarrow$ `RootFinder` $\longrightarrow$ Approximate root

OOP Problem: Putting It Together

- The two classes should work together:

```
p = Polynomial([1, -4, 0, 1])
solver = RootFinder(p)
root = solver.find_root(...)
```

- The `RootFinder` should not need to know how the polynomial stores its coefficients.
- It should simply ask the polynomial to evaluate itself:

```
p.evaluate(x)
```

OOP Problem: Putting It Together

Design

Separate the two responsibilities:

`Polynomial`	`RootFinder`
represents the mathematics	implements the algorithm

Extension: Implement both the Bisection and Newton–Raphson methods in `RootFinder`.